Synthesis versus analysis: what do we actually gain from domain-specificity?

Paul H J Kelly
Group Leader, Software Performance Optimisation
Co-Director, Centre for Computational Methods in Science and Engineering
Department of Computing, Imperial College London

Joint work with:

David Ham (Imperial Computing/Maths/Grantham Inst for Climate Change)
Gerard Gorman, Michael Lange (Imperial Earth Science Engineering – Applied Modelling and Computation Group)
Mike Giles, Gihan Mudadige, Istvan Reguly (Mathematical Inst, Oxford)
Doru Bercea, Fabio Luporini, Graham Markall, Lawrence Mitchell, Florian Rathgeber, Francis Russell, George Rokos,
Paul Colea (Software Perf Opt Group, Imperial Computing)
Spencer Sherwin (Aeronautics, Imperial), Chris Cantwell (Cardio-mathematics group, Mathematics, Imperial)
Michelle Mills Strout, Chris Krieger, Cathie Olschanowsky (Colorado State University)
Carlo Bertolli (IBM Research), Ram Ramanujam (Louisiana State University)
Doru Thom Popovici, Franz Franchetti (CMU), Karl Wilkinson (Capetown), Chris–Kriton Skylaris (Southampton)
This talk is about the following idea:

- can we simultaneously raise the level at which programmers can reason about code,
- provide the compiler with a model of the computation that enables it to generate faster code than you could reasonably write by hand?
This talk is about the following idea:

- can we simultaneously raise the level at which programmers can reason about code,

- provide the compiler with a model of the computation that enables it to generate faster code than you could reasonably write by hand?
Compilation is like skiing

Analysis is not always the interesting part....

Compilation is like skiing

Analysis is not always the interesting part....
What we are doing...

Projects

- **PyOP2/OP2**
  - Unstructured-mesh stencils

- **Firedrake**
  - Finite-element assembly

- **SLAMBench**
  - Dense SLAM – 3D vision

- **PRAgMaTiC**
  - Dynamic mesh adaptation

- **GiMMiK**
  - Small-matrix multiplication

- **TINTL**
  - Fourier interpolation

Contexts

- **Finite-volume CFD**

- **Finite-element**

- **Real-time 3D scene understanding**

- **Adaptive-mesh CFD**

- **Unsteady CFD - higher-order flux-reconstruction**

- **Ab-initio computational chemistry (ONETEP)**

Technologies

- **Vectorisation, parametric polyhedral tiling**

- **Tiling for unstructured-mesh stencils**

- **Lazy, data-driven compute-communicate**

- **Runtime code generation**

- **Multicore graph worklists**

- **Massive common sub-expressions**

- **Optimisation of composite transforms**

Applications

- **Aeroengine turbo-machinery**

- **Weather and climate**

- **Domestic robotics, augmented reality**

- **Tidal turbines**

- **Formula-1, UAVs**

- **Solar energy, drug design**

Targetting MPI, OpenMP, OpenCL, Dataflow/FPGA, from supercomputers to mobile, embedded and wearable
Some examples of domain-specific optimisations

- BLINK: visual effects filters – fusion, vectorisation, CUDA
- DESOLA: runtime fusion for linear algebra
- OP2: (among many) staging for CUDA shared memory
- PyOP2: (ditto) fusion and tiling for unstructured meshes
- COFFEE: (ditto) generalised loop-invariant code motion
- GiMMiK: tiling & full unrolling for block-panel matrix multiply
- TINTL: Fourier interpolation for density functional theory

This talk’s question:

What do we actually gain by building domain-specific tools? Where does the advantage come from?
The standard DSL message:
- Avoid analysis for transformational optimisation
- Transform at the right level of abstraction
- Get the abstraction right

But what do we actually gain by building domain-specific compiler tools?
Unstructured meshes require pointers/indirection because adjacency lists have to be represented explicitly.

A controlled form of pointers.

**OP2** is a C++ and Fortran library for parallel loops over the mesh implemented by source-to-source transformation.

**PyOP2** is a major extension implemented in Python using runtime code generation.

Generates highly-optimised CUDA, OpenMP and MPI code.
**HYDRA: Full-scale industrial CFD using OP2**

- **Unmodified Fortran OP2 source code exploits inter-node parallelism using MPI, and intra-node parallelism using OpenMP and CUDA**
- **Application is a proprietary, full-scale, in-production fluids dynamics package**
- **Developed by Rolls Royce plc and used for simulation of aeroplane engines**

(joint work with Mike Giles, Istvan Reguly, Gihan Mudalige at Oxford)

"Performance portability"

---

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>GPU</th>
<th>RAM</th>
<th>Interconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>HECToR (Cray XE6)</td>
<td>2x16-core AMD Opteron 6276 (Interlagos) 2.3GHz</td>
<td>32GB</td>
<td>128</td>
<td>Cray Gemini</td>
</tr>
<tr>
<td>Jade (NVIDIA GPU Cluster)</td>
<td>2x Tesla K20m +</td>
<td>5GB/GPU (ECC on)</td>
<td>8</td>
<td>FDR InfiniBand</td>
</tr>
</tbody>
</table>

- O3 -h fp3 -h ipa5
- -O2 -xAVX
- -arch=sm_35 -use_fast_math

---

**Fig. 9:**
(a) Strong Scaling (2.5M edges)
(b) Weak Scaling (0.5M edges per node)
Where did the domain-specific advantage come from?

- Automatic code synthesis, for MPI, OpenMP, CUDA, OpenCL – single source code, clean API
- Inspector-executor scheme: we know we will iterate over the mesh many times, so we can invest in partitioning, colouring etc
- Code synthesis templates to deliver optimised implementations, eg:
  - Colouring to avoid read-increment-write conflicts
  - Staging of mesh data into CUDA shared memory
  - Splitting push loops (that increment via a map) to reduce register pressure (LCPC2012)
**Sparse split tiling** on an unstructured mesh, for locality

---

**Loop 1**
- Visits edges
- Increments nodes

**Loop 2**
- Visits nodes
- Depends on edges

---

How can we fuse two loops, when there is a “halo” dependence?

I.e. load a block of mesh and do the iterations of loop 1, then the iterations of loop 2, before moving to the next block.

If we could, we could dramatically improve the memory access behaviour!
Sparse split tiling

Partition the iteration space of loop 1

Loop 1
- Visits edges
- Increments nodes

Loop 2
- Visits nodes
- Depends on edges

Strout, Luporini et al, IPDPS'14
Partition the iteration space of loop 1

- Colour the partitions
- Project the tiles, using the knowledge that colour n can use data produced by colour n-1

Thus, the tile coloured #1 grows where it meets colour #0

And shrinks where it meets colours #2 and #3
OP2 loop fusion in practice

Mesh size = 1.5M edges
## Loop chain = 6 loops
No inspector/plans overhead

Airfoil test problem
Unstructured-mesh finite-volume

![Graph showing speedup of Airfoil on Sandy Bridge](image-url)

Intel Sandy Bridge (dual-socket 8-core Intel Xeon E5-2680 2.00Ghz, 20MB of shared L3 cache per socket); Intel icc 2013 (-O3, xSSE4.2/-xAVX).
Where did the domain-specific advantage come from?

- OP2’s access descriptors provide precise dependence iteration-to-iteration information
- We “know” that we will iterate many times over the same mesh – so it’s worth investing in an expensive “inspector-executor” scheme

- We capture chains of loops over the mesh
  - We could get our compiler to find adjacent loops
  - We could extend the OP2 API with “loop chains”

What we actually do?

- We delay evaluation of parallel loops
- We build a chain (DAG) of parallel loops at runtime
- We generate code at runtime for the traces that occur
The finite element method in outline

\[
\text{do element} = 1,N \\
\text{assemble(element)} \\
\text{end do}
\]

\[
\int_{\Omega} vL(u^\delta) dX = \int_{\Omega} vq dX.
\]

\[
Ax = b
\]

Key data structures: Mesh, dense local assembly matrices, sparse global system matrix, and RHS vector
Multilayered abstractions for FE

Local assembly:
- Specified using the FEniCS project’s DSL, UFL (the “Unified Form Language”)
- Computes local assembly matrix
- Key operation is evaluation of expressions over basis function representation of the element

Mesh traversal:
- OP2
- Loops over the mesh
- Key is orchestration of data movement

Solver:
- Interfaces to standard solvers, such as PetSc
A weak form of the shallow water equations

\[ \int_{\Omega} q \nabla \cdot \mathbf{u} dV = - \int_{\Gamma_E} \mathbf{u} \cdot \mathbf{n}(q^{+} - q^{-}) dS \]

\[ \int_{\Omega} \mathbf{v} \cdot \nabla h dV = c^2 \int_{\Gamma_E} (h^{+} - h^{-}) \mathbf{n} \cdot \mathbf{v} dS \]

can be represented in UFL as

**UFL source**

```python
V = FunctionSpace(mesh, 'Raviart-Thomas', 1)
H = FunctionSpace(mesh, 'DG', 0)
W = V*H
(v, q) = TestFunctions(W)
(u, h) = TrialFunctions(W)
M_u = inner(v, u)*dx
M_h = q*h*dx
Ct = -inner(avg(u), jump(q, n))*dS
C = c**2*adjoint(Ct)
F = f*inner(v, as_vector([-u[1], u[0]]))*dx
A = assemble(M_u + M_h + 0.5*dt*(C-Ct+F))
A_r = M_u + M_h - 0.5*dt*(C-Ct+F)
```

**Local assembly kernel**

```python
void Mass(double localTensor[3][3])
{
    const double qw[6] = { ... };
    const double CG1[3][6] = { ... };
    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++)
            for(int g = 0; g < 6; g++)
                localTensor[i][j] += CG1[i][g] * CG1[j][g] * qw[g];
}
```

parallel loop

over all grid cells,
in unspecified order,
partitioned

unstructured grid
defined by vertices, edges and cells
**Firedrake: a finite-element framework**

- An alternative implementation of the FEniCS language
- Using PyOP2 as an intermediate representation of parallel loops
- All embedded in Python

- The FEniCS project’s UFL – DSL for finite element discretisation
- Compiler generates PyOP2 kernels and access descriptors
- Stencil DSL for *unstructured-mesh*
- Explicit *access descriptors* characterise access footprint of kernels
- Runtime code generation
The advection-diffusion problem:

\[ \frac{\partial T}{\partial t} = D \nabla^2 T - \mathbf{u} \cdot \nabla T \]

Weak form:

\[ \int_\Omega q \frac{\partial T}{\partial t} \, dX = \int_{\partial\Omega} q(\nabla T - \mathbf{u} T) \cdot \mathbf{n} \, ds - \int_\Omega \nabla q \cdot \nabla T \, dX + \int_\Omega \nabla q \cdot \mathbf{u} T \, dX \]

This is the entire specification for a solver for an advection-diffusion test problem.

Same model implemented in FEniCS/Dolfin, and also in Fluidity – hand-coded Fortran.

```python
t=state.scalar_fields["Tracer"]  # Extract fields
u=state.vector_fields["Velocity"]  # from Fluidity
p=Trialfunction(t)  # Setup test and
q=Testfunction(t)  # trial functions
M=p*q*dx  # Mass matrix
d=-dt*dfsuty*dot(grad(q),grad(p))*dx  # Diffusion term
D=M-0.5*d  # Diffusion matrix
adv = (q*t+dt*dot(grad(q),u)*t)*dx  # Advection RHS
diff = action(M+0.5*d,t)  # Diffusion RHS
solve(M == adv, t)  # Solve advection
solve(D == diff, t)  # Solve diffusion
```
Here we compare performance against two production codes solving the same problem on the same mesh:

- Fluidity: Fortran/C++
- DOLFIN: the FEniCS project’s implementation of UFL

These results are preliminary and are presented for discussion purposes – see Rathgeber, Ham, Mitchell et al, http://arxiv.org/abs/1501.01809 for more systematic evaluation.

Graph shows speedup over Fluidity on one core of a 12-core Westmere node.
Where did the domain-specific advantage come from?

- UFL (the Unified Form Language, inherited from the FEniCS Project)
  - Delivers spectacular expressive power
  - Reduces scope for coding errors
  - Supports flexible exploration of different models, different PDEs, different solution schemes

- Building on PyOP2
  - Handles MPI, OpenMP, CUDA, OpenCL
  - Completely transparently

- PyOP2 uses runtime code generation
  - So we don’t need to do static analysis
  - So the layers above can freely exploit unlimited abstraction
Where did the domain-specific advantage come from?

The adjoint of the PDE characterises the sensitivity of the PDE’s solution to input variables; it is usually derived by automatic differentiation of the solver code:

\[ \text{discrete forward equations} \xrightarrow{\text{implement model by hand}} \text{forward code} \]

\[ \text{algorithmic differentiation} \xrightarrow{} \text{adjoint code} \]

With UFL we have access to the PDE so we can generate the adjoint solver directly:

\[ \text{discrete forward equations} \xrightarrow{\text{FEniCS/Firedrake}} \text{forward code} \]

\[ \text{libadjoint} \xrightarrow{} \text{discrete adjoint equations} \xrightarrow{\text{FEniCS/Firedrake}} \text{adjoint code} \]
Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange $p = 1$ elements.

The local assembly operation computes a small dense submatrix essentially computing (for example) integrals of flows across facets.

These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE.
Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange $p = 1$ elements.

The local assembly operation computes a small dense submatrix essentially computing (for example) integrals of flows across facets. These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE.
void helmholtz(double A[3][4], double **coords) {
    #define ALIGN __attribute__((aligned(32)))
    // K, det = Compute Jacobian (coords)

    static const double W[3] ALIGN = { ... }
    static const double X_D10[3][4] ALIGN = { { ... } }
    static const double X_D01[3][4] ALIGN = { { ... } }

    for (int i = 0; i<3; i++) {
        double LI_0[4] ALIGN;
        double LI_1[4] ALIGN;
        for (int r = 0; r<4; r++) {
            LI_0[r] = ((K1*X_D10[i][r])+(K3*X_D01[i][r]));
            LI_1[r] = ((K0*X_D10[i][r])+(K2*X_D01[i][r]));
        }
        for (int j = 0; j<3; j++)
            #pragma vector aligned
            for (int k = 0; k<4; k++)
                A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j]+LI_1[k]*LI_1[j])*det*W[i]);
    }
}
void burgers(double A[12][12], double **coords, double **w) 
// K, det = Compute Jacobian (coords)

static const double W[5] = {...}
static const double X1_D001[5][12] = {{...}}
static const double X2_D001[5][12] = {{...}}
//11 other basis functions definitions.

... 
for (int i = 0; i<5; i++) {
    double F0 = 0.0;
    //10 other declarations (F1, F2,...)
    ...
    for (int r = 0; r<12; r++) {
        F0 += (w[r][0]*X1_D100[i][r]);
        //10 analogous statements (F1, F2, ...)
    }
    ...
    for (int j = 0; j<12; j++)
        for (int k = 0; k<12; k++)
            A[j][k] += ((K5*F9)+(K8*F10))*Y1[i][j]+
            +((K0*X1_D100[i][k])+(K3*X1_D010[i][k])+(K6*X1_D001[i][k]))*Y2[i][j])*F11)+
            +((K2*X2_D100[i][k])+...+(K8*X2_D001[i][k]))*((K2*X2_D100[i][j])...+(K8*X2_D001[i][j])..)+
            + <roughly a hundred sum/muls go here>).*
            *det*W[i];
}
Fairly serious, realistic example: static linear elasticity, $p=2$
tetrahedral mesh, 196608 elements
Including both assembly time and solve time
Single core of Intel Sandy Bridge
Compared with Firedrake loop nest compiled with Intel’s icc compiler version 13.1
At low $p$, matrix insertion overheads dominate assembly time
At higher $p$, and with more coefficient functions ($f=2$), we get up to 1.47x overall application speedup
Where did the domain-specific advantage come from?

- Finite-element assembly kernels have complex structure
- With rich loop-invariant expression structure
- And simple dependence structure

COFFEE generates C code that we feed to the best available compiler

COFFEE’s transformations make this code run faster

COFFEE does not use any semantic information not available to the C compiler

- But it does make better decisions
- For the loops we’re interested in
Where did the domain-specific advantage come from?

COFFEE does “generalised” loop-invariant code motion (GLICM)

“an expression in a loop \( L \) is invariant with respect to a parent loop \( P \) if each of its operands is

- defined outside of \( P \),
- or is the induction variable of \( L \),
- or is the induction variable of a superloop of \( L \) which is also a subloop of \( P \).”

We have an implementation for LLVM… preliminary evaluation suggests rather few general C programs benefit from GLICM

```
1 int A[100];
2 int x=0, y=0;
3 int t1[100];
4 for (int j=0; j<100; j++) {
  5     t1[j]=A[j][n-j]*A[n-j][j];
7 }  
8 }
9 for (int i=0; i<100; i++) {
10     x+=t2;
11     y+=t1;
12 }  
13 }  
```

(a) Original loop nest.

```
1 int A[100];
2 int x=0, y=0;
3 int t1[100];
4 for (int j=0; j<100; j++) {
  5     t1[j]=A[j][n-j]*A[n-j][j];
7 }  
8 }
9 for (int i=0; i<100; i++) {
10     x+=t2;
11     y+=t1;
12 }  
13 }
```

(b) After LICM and GLICM.

Figure 4.8: A loop nest where loop interchanging and LICM hoist a suboptimal number of invariant expressions.

Even for programs such as the one in Figure 4.7, where we can use both methods to hoist the same number of invariant instructions, the GLICM-optimized version might perform better in practice. Firstly, the cloned loop introduced by GLICM is vectorizable. If automatic vectorization is enabled, the hoisted operations will be executed using SIMD instructions, achieving a speed-up compared to the scalar versions. Secondly, loop interchanging might have other detrimental effects on the runtime of the program, such as increasing cache misses due to inefficient memory access patterns.
Conclusions

Where do DSO opportunities come from?
- Domain semantics (eg in SPIRAL)
- Domain expertise (eg we know that inspector-executor will pay off)
- Domain idiosyncracies (eg for GLICM)
- Transforming at the right representation
  - Eg fusing linear algebra ops instead of loops
- Data abstraction (eg AoS vs SoA)
  - Or whether to build the global system matrix (in instead to use a matrix-free or local-assembly scheme)

Runtime code generation is liberating
- We do not try to do static analysis on client code
- We encourage client code to use powerful abstractions
Partly funded by

- NERC Doctoral Training Grant (NE/G523512/1)
- EPSRC “MAPDES” project (EP/I00677X/1)
- EPSRC “PSL” project (EP/I006761/1)
- Rolls Royce and the TSB through the SILOET programme
- EPSRC “PAMELA” Programme Grant (EP/K008730/1)
- EPSRC “PRISM” Platform Grant (EP/I006761/1)
- EPSRC “Custom Computing” Platform Grant (EP/I012036/1)
- AMD, Codeplay, Maxeler Technologies

Code:

- http://www.firedrakeproject.org/
- http://op2.github.io/PyOP2/
Welcome to PyOP2’s documentation!

Contents:

- Installing PyOP2
  - Quick start
  - Provisioning a virtual machine
  - Preparing the system
  - Dependencies
  - Building PyOP2
  - Setting up the environment
  - Testing your installation
  - Troubleshooting

- PyOP2 Concepts
  - Sets and mappings
  - Data
  - Parallel loops

- PyOP2 Kernels
  - Kernel API
  - Data layout
  - Local iteration spaces

- The PyOP2 Intermediate Representation
  - Using the Intermediate Representation
  - Achieving Performance Portability with the IR
  - Optimizing kernels on CPUs
  - How to select specific kernel optimizations

- PyOP2 Architecture
  - Multiple Backend Support
Firedrake is an automated system for the portable solution of partial differential equations using the finite element method (FEM). Firedrake enables users to employ a wide range of discretisations to an infinite variety of PDEs and employ either conventional CPUs or GPUs to obtain the solution.

Firedrake employs the Unified Form Language (UFL) and FEniCS Form Compiler (FFC) from the FEniCS Project and fields and meshes from Fluidity. The parallel execution of the FEM solver is accomplished by the PyOP2 system.

- The Firedrake team
  - Summer students 2013
- Obtaining Firedrake
  - PyOP2
  - Firedrake
The FEniCS project... The book

Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book (Lecture Notes in Computational Science and Engineering) [Hardcover]
Anders Logg (Editor), Kent-Andre Mardal (Editor), Garth Wells (Editor)
Be the first to review this item

Price: £62.99 & this item Delivered FREE in the UK with Super Saver Delivery. See details and conditions

In stock but may require up to 2 additional days to deliver.
Dispatched from and sold by Amazon. Gift-wrap available.

32 new from £46.11  8 used from £51.23
computer science is a science of abstraction — creating the right model for thinking about a problem and devising the appropriate mechanizable techniques to solve it